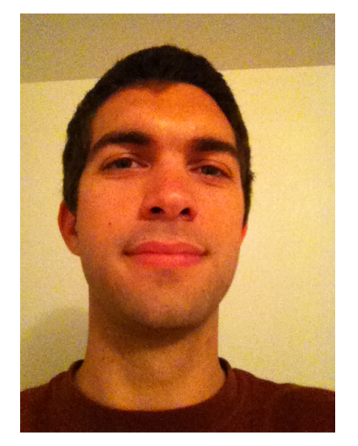
Graph Pattern Matching With Hadoop Giraph

John Gilmer, Mentor – Xinh Huynh (GS)



In our work we have leveraged capabilities of Apache's open source software framework, Giraph, to implement a graph pattern matching algorithm. Graphs can be processed intuitively through message passing between vertices along edges. The software framework Giraph encapsulates this idea and then runs the algorithm in parallel on top of the Hadoop distributed computing framework to achieve quicker processing times. Our work included leveraging the API of Hadoop Giraph to run custom algorithms such as the Subgraph Isomorphism Matching Algorithm. The next step in this project is to link the Giraph implementation to the Accumulo database and run the algorithm on non-trivial data sets to measure performance.

Introduction

Advantages of Giraph Implementation

◆ Giraph is based on Google's Pregel model (figure 1) which has a speed advantage over MapReduce graph algorithms

◆ Open source software is free of use and is being constantly improved by the community

◆ Giraph is built on Hadoop so it can process very large graphs (Facebook ran a Giraph algorithm on a 1 billion edge graph)

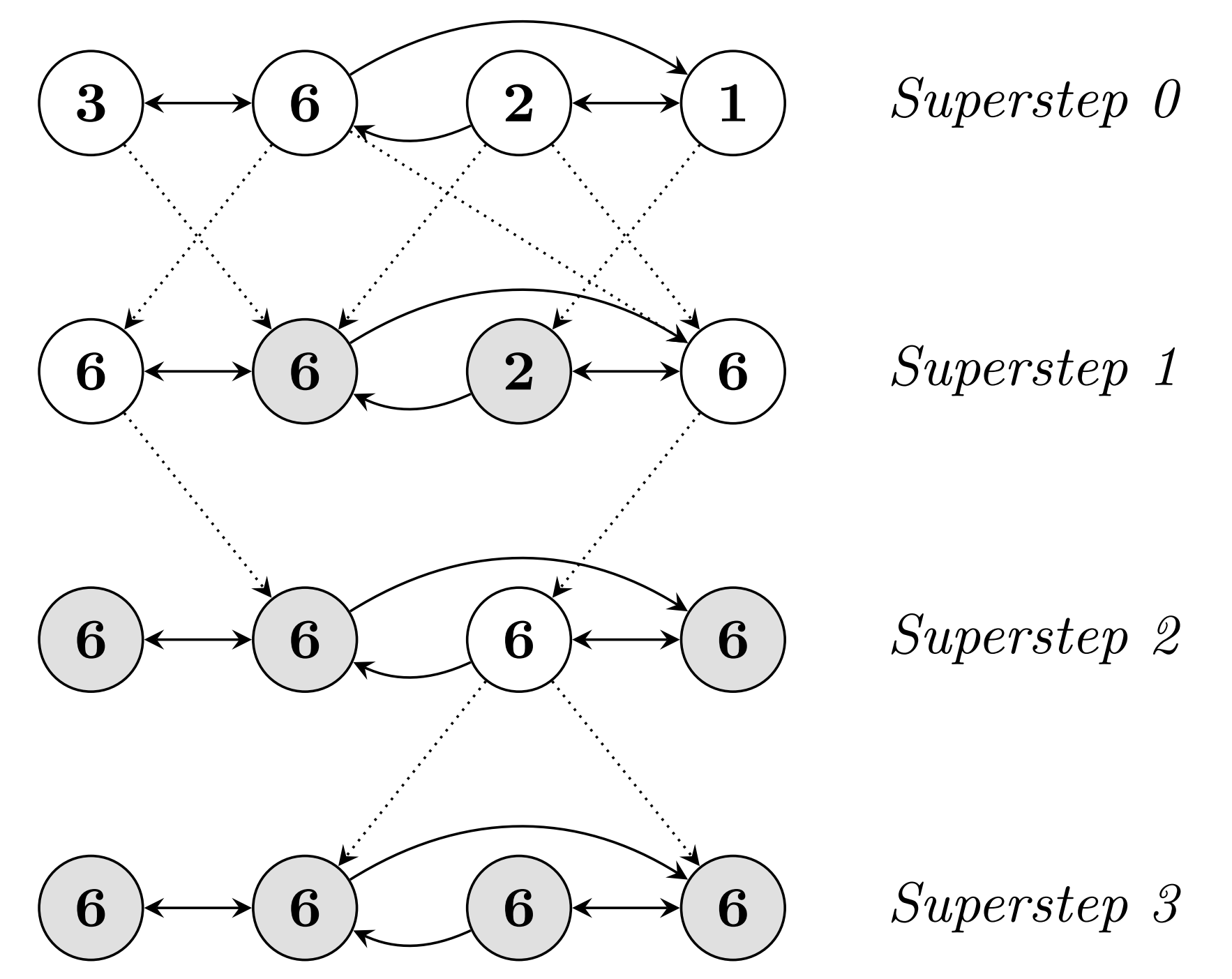


Figure 1: Pregel Algorithm for Largest Value

Process of a Superstep

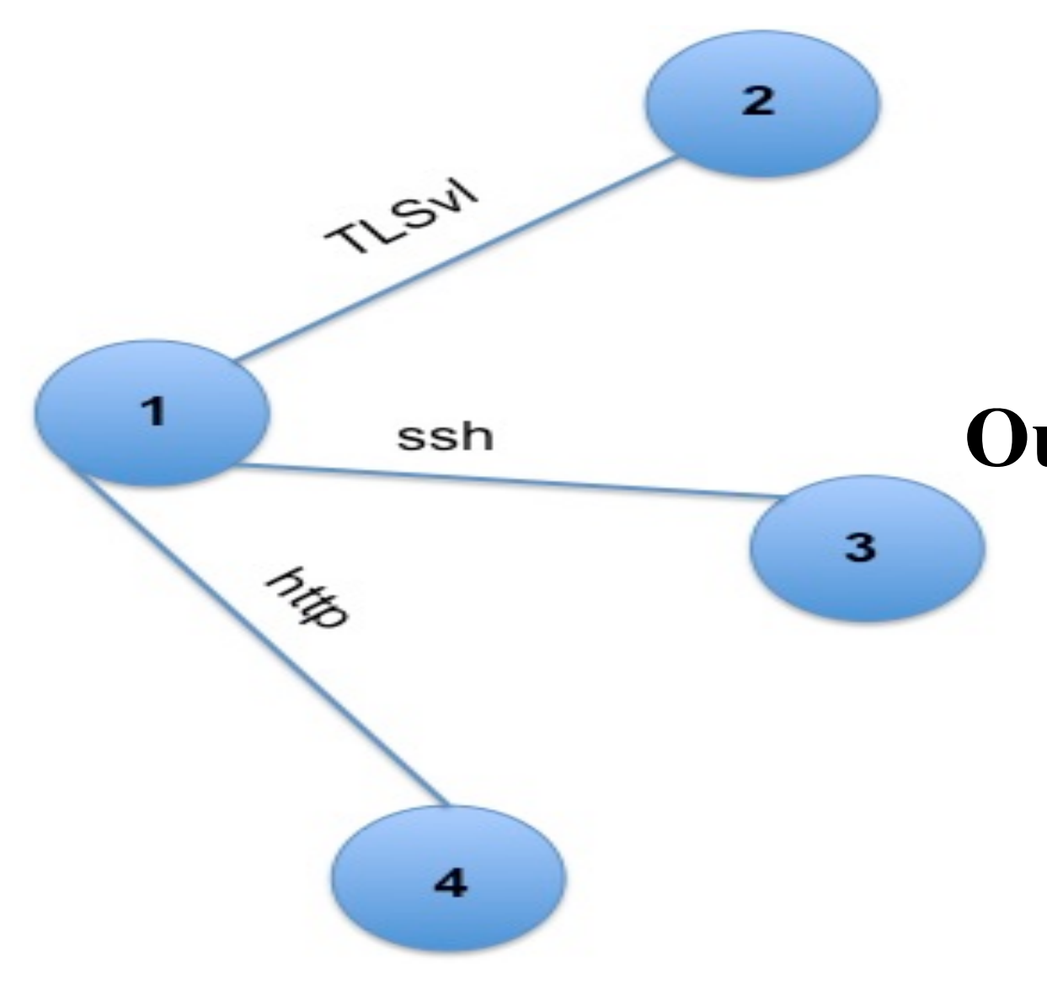
◆ Run user-defined compute function on the vertex

◆ Read messages sent to it from the last superstep

◆ Send messages to other vertices

◆ Modify state of graph

Subgraph Isomorphism Matching Algorithm



Input: Graph G, Walk W See Figure 2.

Output: Set of matching sub-graphs within G.

Figure 2: Example of a subgraph to be matched. Must be converted to a walk for the algorithm to work.

Methods

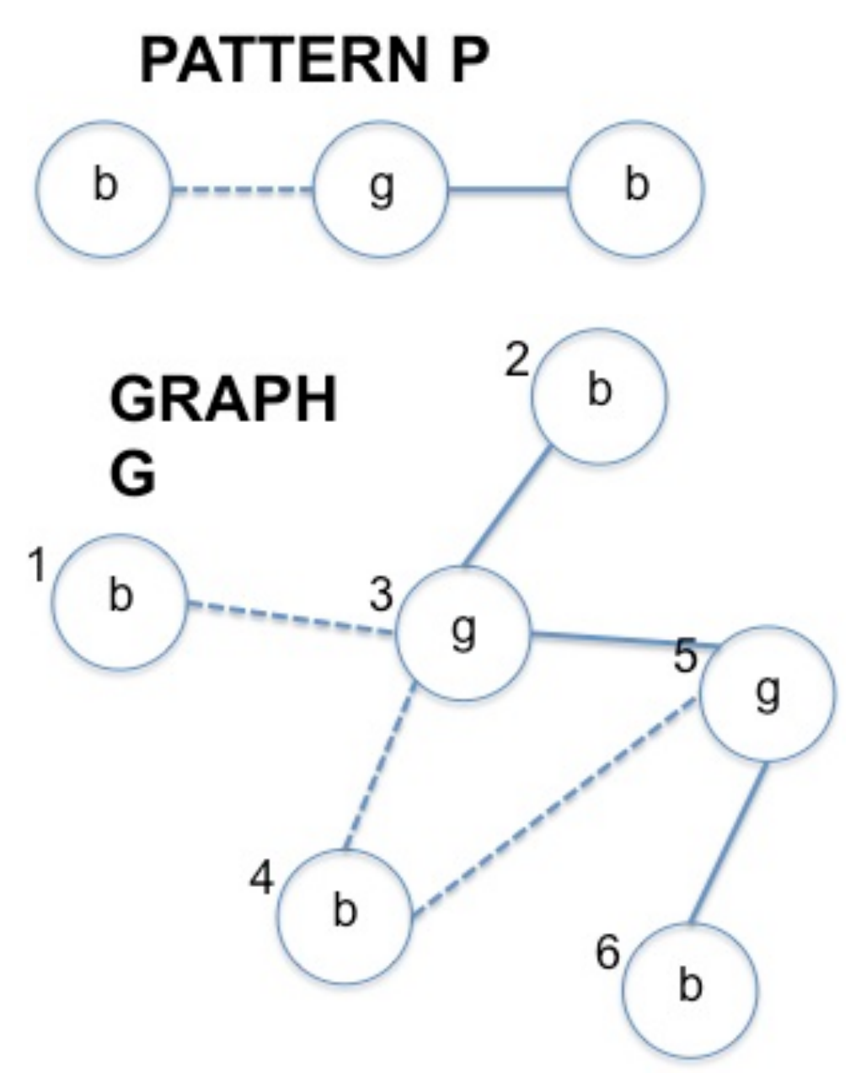
Step 1: First we examined Apache's documentation and code to understand how the software worked and to plan our implementation. With their provided examples we implemented a simple shortest paths algorithm.

Important Giraph Classes			
Vertex	**Input Format**	**Ouptut Format**	**GraphJob**
Compute Function and Message Handling	Builds vertices on worker from structured Input File	Takes graph data from worker and outputs it	Set Giraph configurations and sends job to Map Reduce

Step 2: Use knowledge of software structure to reformat the program to run the subgraph matching algorithm.

Comparing Algorithms	
Algorithm	**Computation**
Simple Shortest Path Algorithm	1. Read Graph 2. Change Graph Values 3. Output Graph
Subgraph Isomorphism Matching Algorithm	1. Read Graph 2. Record Set of Subgraphs 3. Output Set

Giraph was set up to do computation processes that are more similar to that of the Simple Shortest Paths Algorithm than the Subgraph Isomorphism Matching Algorithm. To run this slightly different computation process on Giraph we would have to take an alternate route.



Desired Output

1 ---> 3 ---> 2
4 ---> 3 ---> 2
4 ---> 5 ---> 6

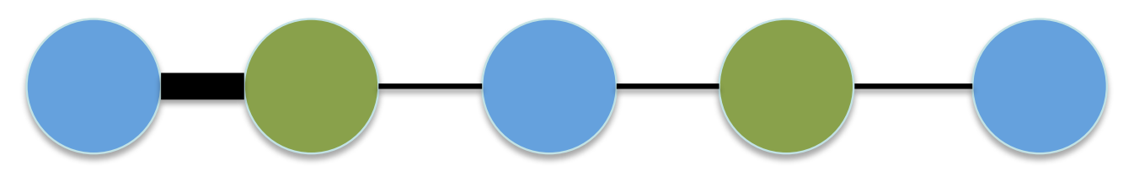
Pattern: The pattern of vertex and edge types must be stored somewhere. This additional constraint is handled by the GraphJobWithPattern class which we wrote to implement a pattern as a configuration variable.

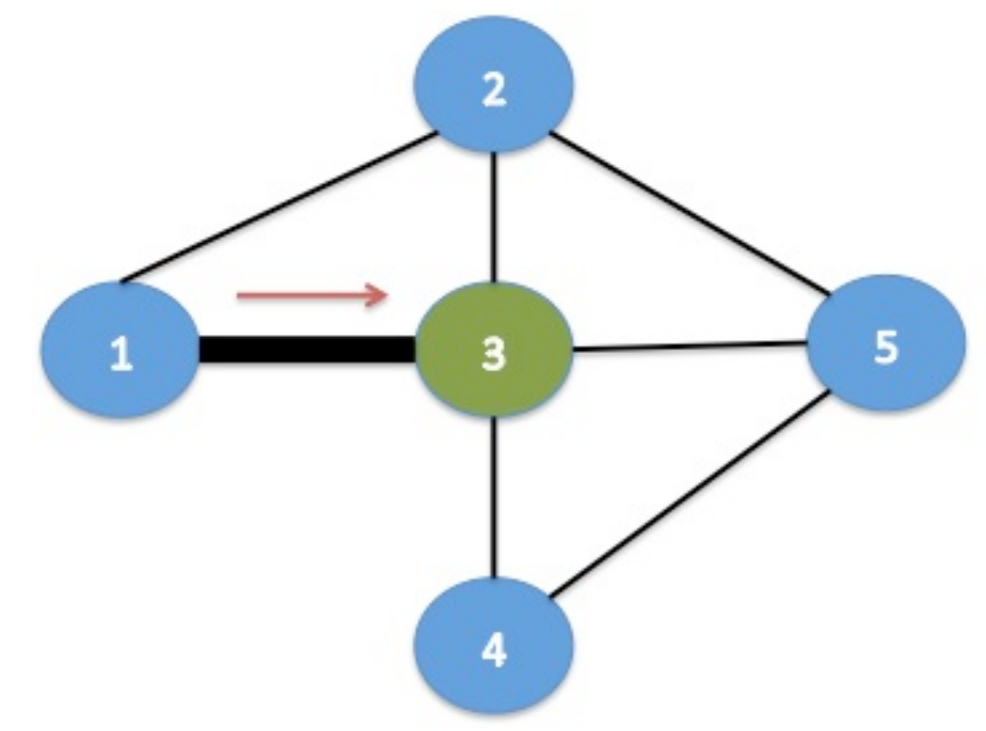
Graph: The graph does not change at all so the compute function simply waits till a vertex says it knows a matching walk and then it tells WorkerContext class to keep track of all the matching walks from the worker.

Output: Normally in a Giraph program a client writes an output class and links it to GraphJob. GraphJob then automatically calls it on the graph. Here we do not want to output the graph, we want a subset of the graph and so we output from the WorkerContext class.

Results

Input Pattern P:





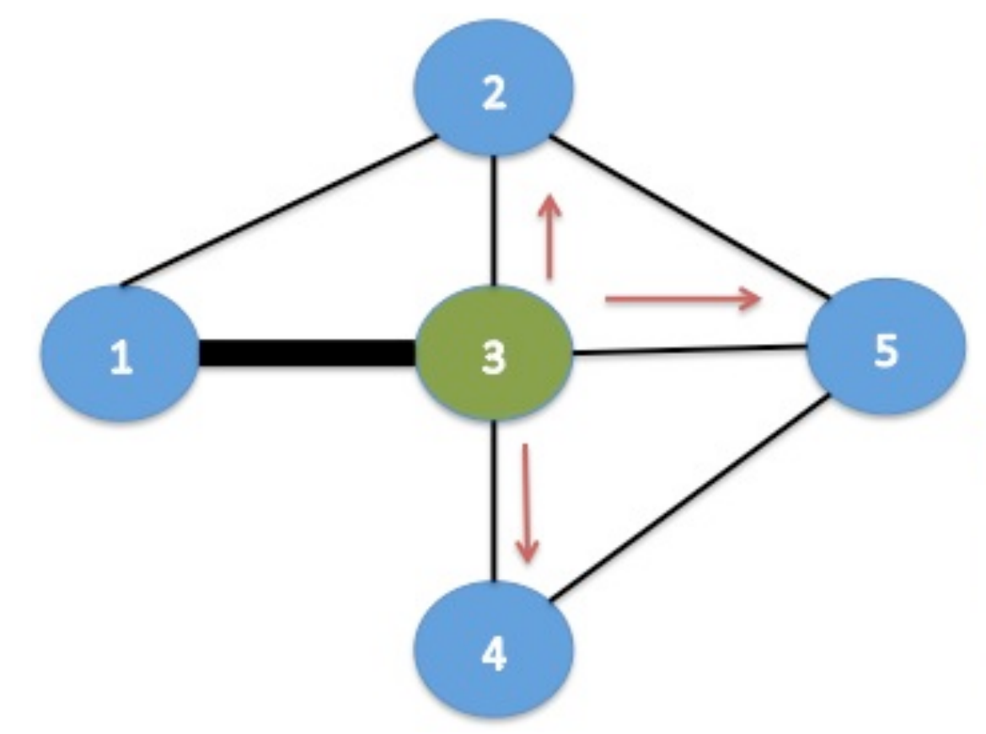
Superstep 0

Messages Received:

None

Messages Sent:

Source	Destination	Message
1	3	<1,0,0,0,0>



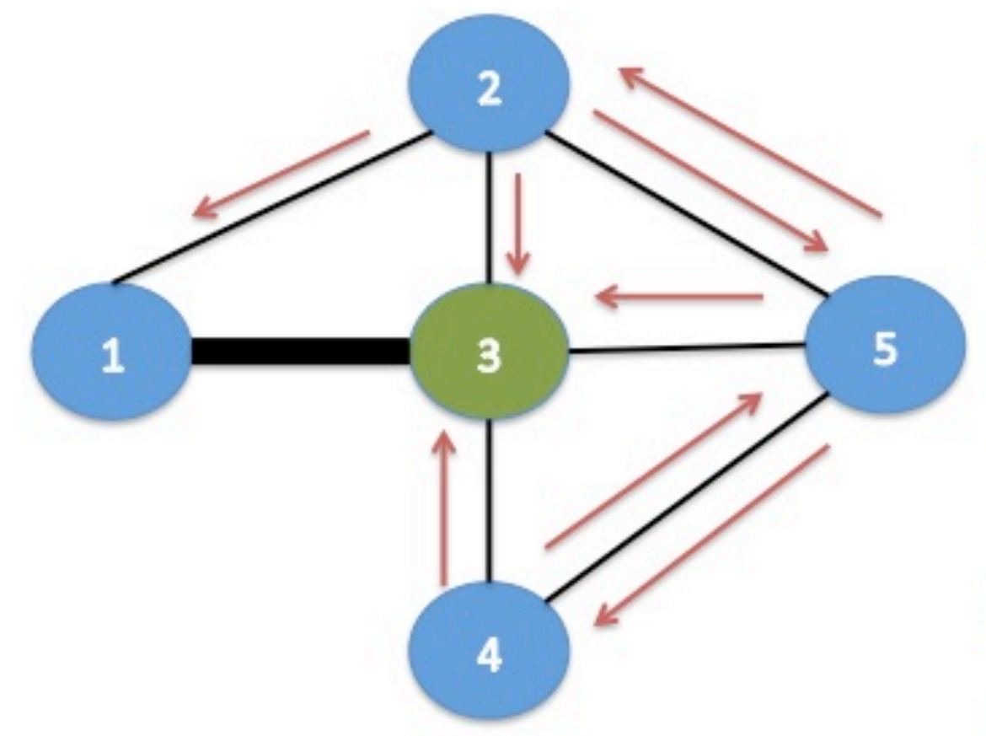
Superstep 1

Messages Received:

Source	Destination	Message
1	3	<1,0,0,0,0>

Messages Sent:

Source	Destination	Message
3	2	<1,3,0,0,0>
3	4	<1,3,0,0,0>
3	5	<1,3,0,0,0>



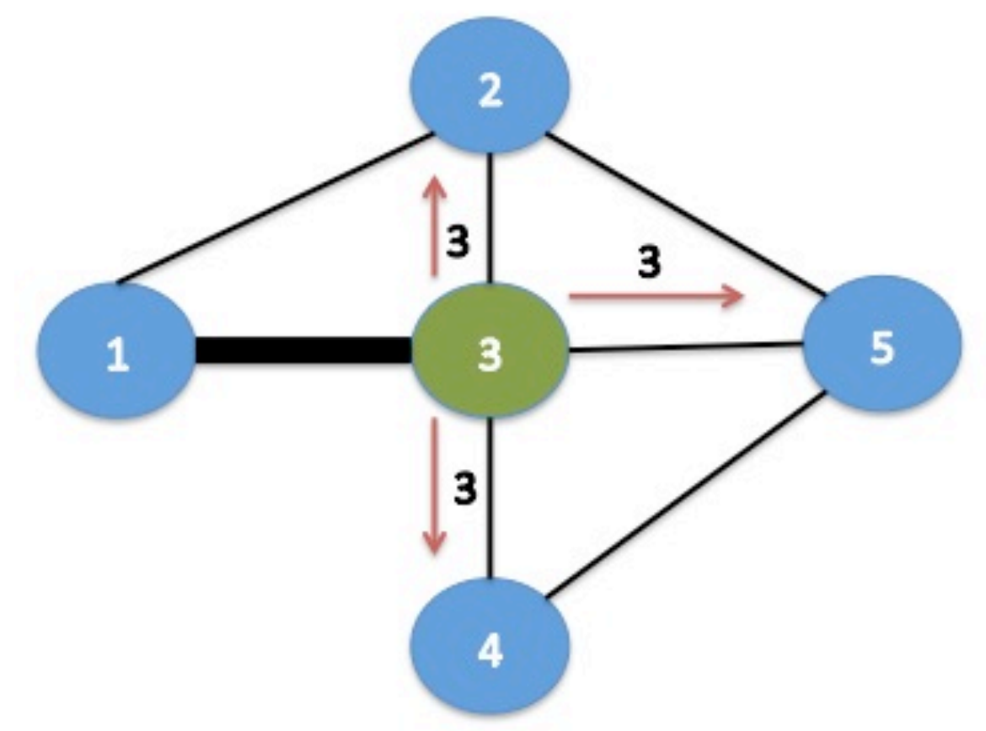
Superstep 2

Messages Received:

Source	Destination	Message
3	2	<1,3,0,0,0>
3	4	<1,3,0,0,0>
3	5	<1,3,0,0,0>

Messages Sent:

Source	Destination	Message
2	1	<1,3,2,0,0>
2	3	<1,3,2,0,0>
2	5	<1,3,2,0,0>
5	2	<1,3,5,0,0>
5	4	<1,3,5,0,0>
5	3	<1,3,5,0,0>
4	3	<1,3,4,0,0>
4	5	<1,3,4,0,0>



Superstep 3

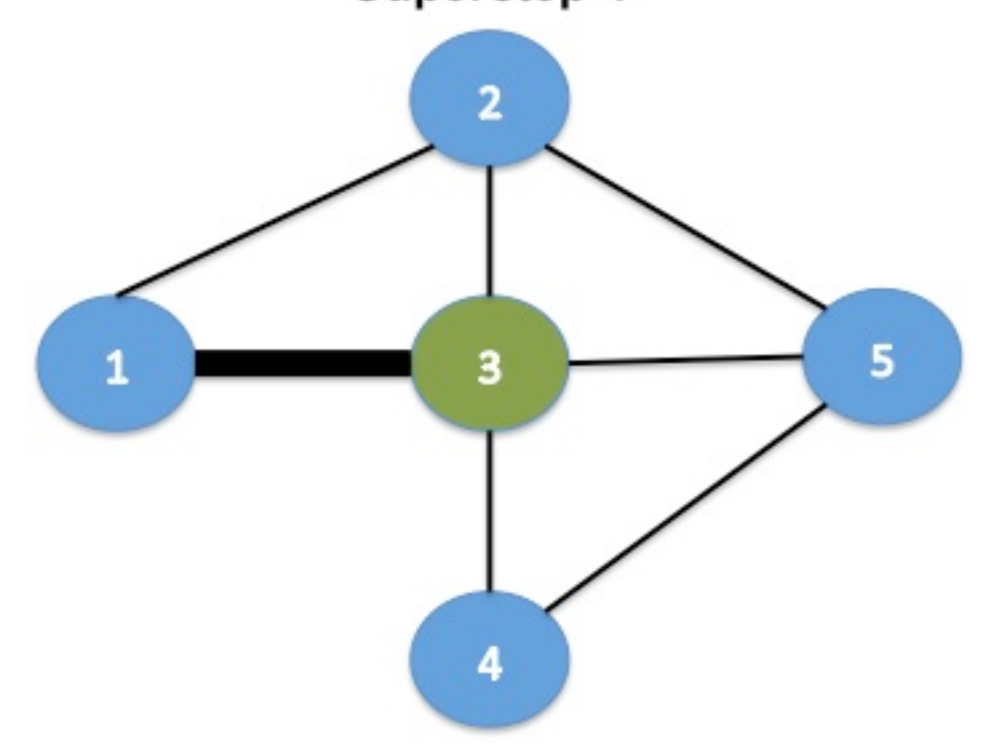
Messages Received That Matched Pattern:

Source	Destination	Message
2	3	<1,3,2,0,0>
4	3	<1,3,4,0,0>
5	3	<1,3,5,0,0>

Messages Sent:

Source	Destination	Message
3	2	<1,3,2,3,0>
3	4	<1,3,2,3,0>
3	5	<1,3,2,3,0>
3	2	<1,3,4,3,0>
3	4	<1,3,4,3,0>
3	5	<1,3,4,3,0>
3	2	<1,3,5,3,0>
3	4	<1,3,5,3,0>
3	5	<1,3,5,3,0>

Superstep 4



Superstep 4 is the final superstep. All the messages sent from superstep 3 will match the vertex type of their destination which is the last step of the pattern. From here the code outputs all 9 matching subgraphs. Superstep 4 sends no messages and all vertices vote to halt.

Next Steps

Implement Accumulo: We need to move from the current flat file implementation to the more practical Accumulo DB implementation as seen in figure 3.

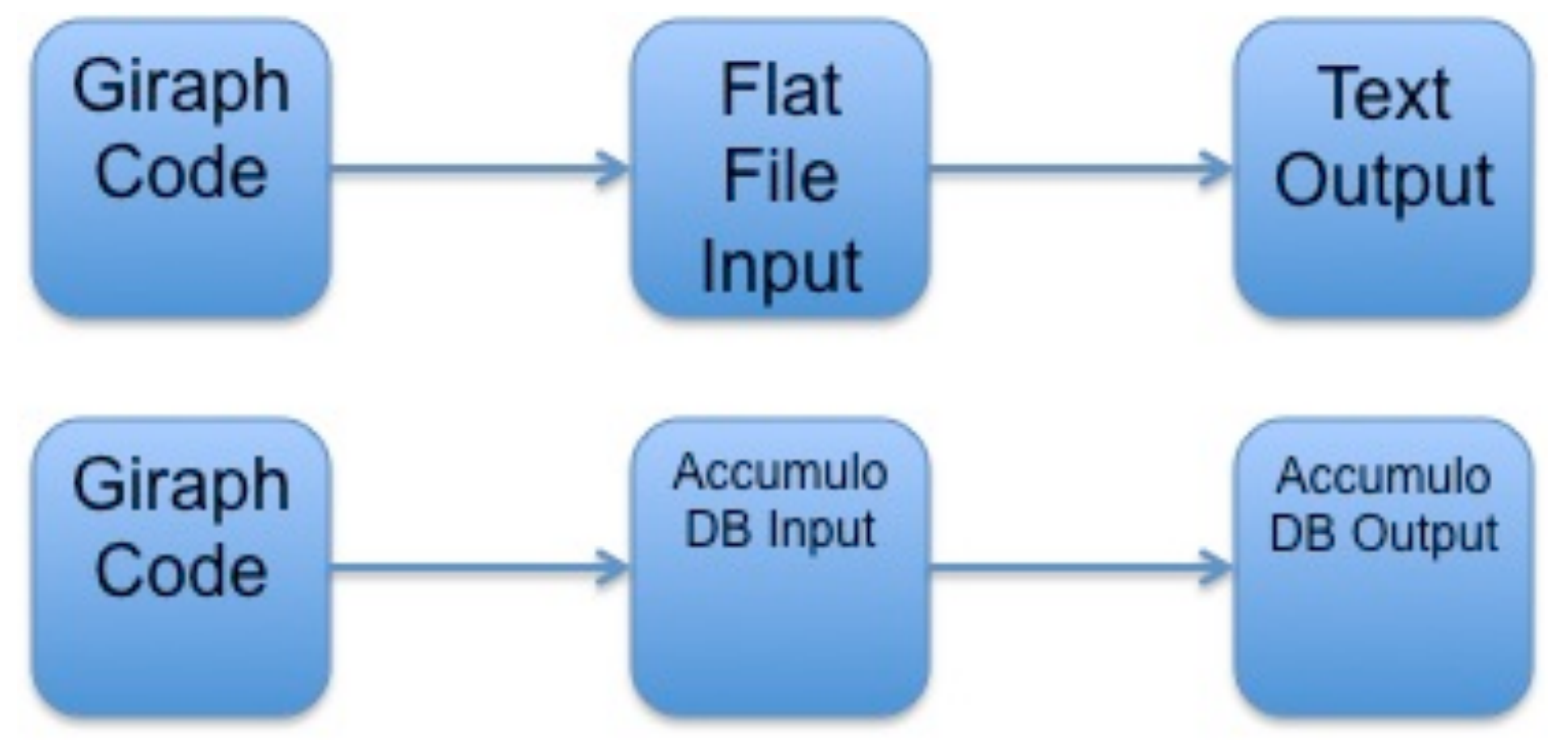


Figure 3: Accumulo DB Implementation

Optimization: The current implementation of the algorithm does not change based on the metrics of the graph. For example if Pattern P (figure 4) exists such that the second vertex type (green) occurs once in the target graph(figure 5) while the first vertex type (blue) occurs 1000000 times then our message passing portion of the algorithm would be very inefficient(figure 6). We would send at least 1000000 messages when we would only need to send 1 (figure 7).

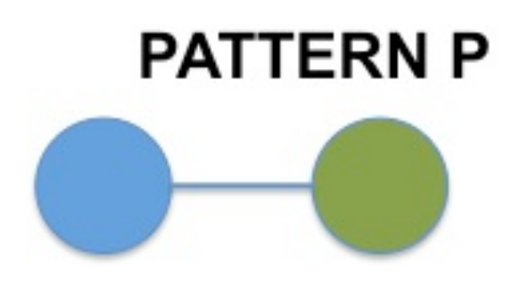


Figure 4: Pattern P of blue node with edge to green node

Graph G Messages

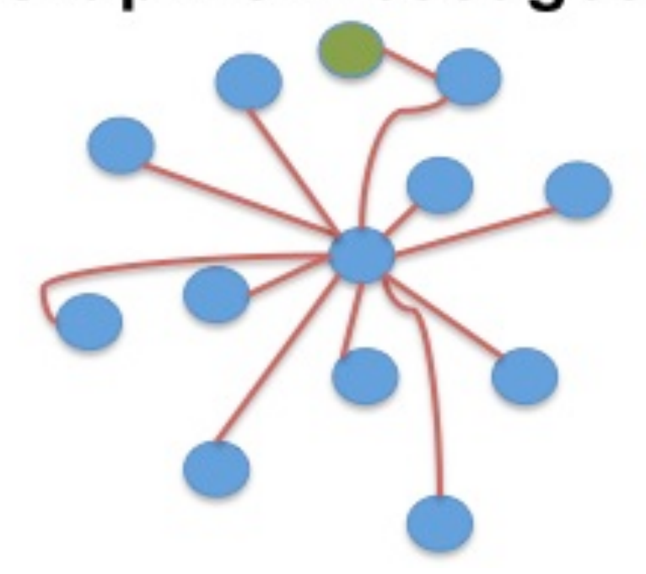


Figure 6: Message passing in current algorithm

Graph G

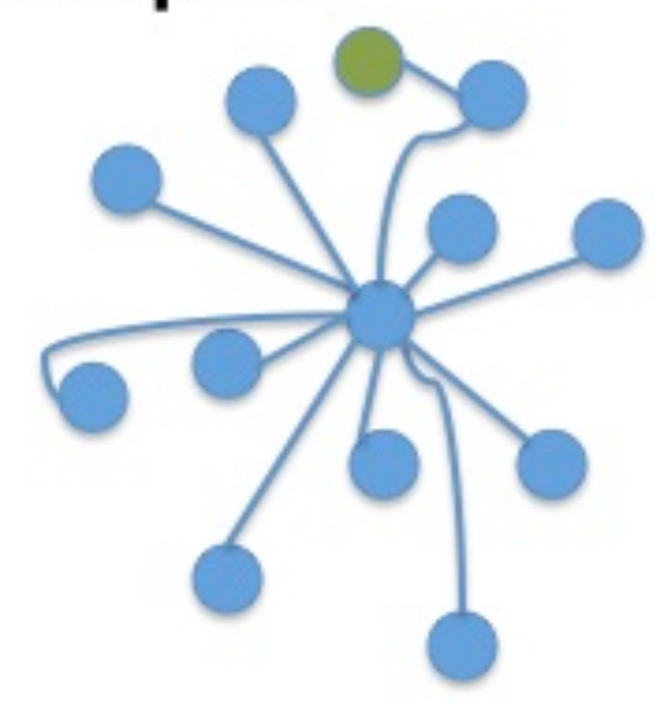


Figure 5: Graph G with many blue nodes and 1 green

Graph G Messages

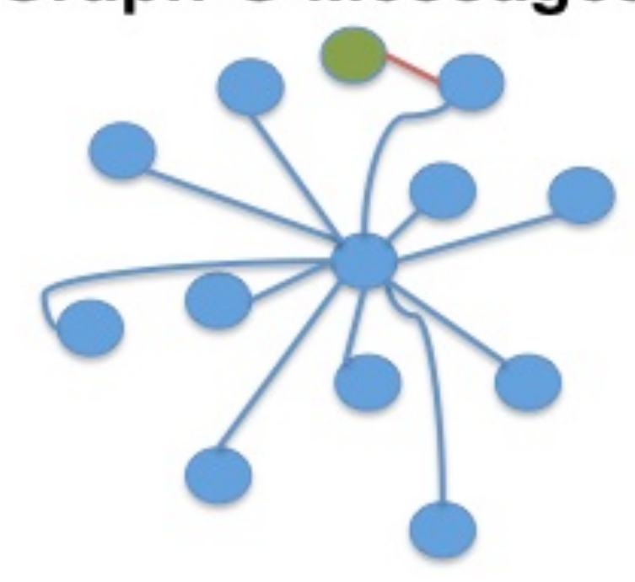


Figure 7: Message passing in optimized algorithm

IM Release: LLNL-POST-571213